Des. Codes Cryptogr. (2010) 56:141–162
DOI 10.1007/s10623-010-9391-y

Whirlwind: a new cryptographic hash function

Paulo Barreto · Ventzislav Nikov · Svetla Nikova · Vincent Rijmen · Elmar Tischhauser

Received: 14 August 2009 / Revised: 24 March 2010 / Accepted: 24 March 2010 /
Published online: 16 April 2010

Abstract A new cryptographic hash function *Whirlwind* is presented. We give the full specification and explain the design rationale. We show how the hash function can be implemented efficiently in software and give first performance numbers. A detailed analysis of the security against state-of-the-art cryptanalysis methods is also provided. In comparison to the algorithms submitted to the SHA-3 competition, Whirlwind takes recent developments in cryptanalysis into account by design. Even though software performance is not outstanding, it compares favourably with the 512-bit versions of SHA-3 candidates such as LANE or the original CubeHash proposal and is about on par with ECHO and MD6.

Keywords Cryptographic hash functions · Whirlpool · Normal bases over finite fields · Dyadic matrices · Rebound attacks

Mathematics Subject Classification (2000) 94A60 · 12E30

Dedicated to the memory of András Gács (1969–2009).

Communicated by L. Storme.

P. Barreto
Departamento de Engenharia de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, São Paulo, Brazil

V. Nikov
NXP Semiconductors, Leuven, Belgium

S. Nikova
EEMCS-DIES, University of Twente, Enschede, The Netherlands

S. Nikova (✉) · V. Rijmen · E. Tischhauser
Department of ESAT/SCD-COSIC and IBBT, Katholieke Universiteit Leuven, Louvain, Belgium
e-mail: svetla.nikova@esat.kuleuven.ac.be

V. Rijmen
Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Graz, Austria

1 Introduction

Whirlwind is a cryptographic hash function that follows the Sponge model [5]. Its compression function is based on the repeated application of a round transformation, similar to a block cipher, and designed according to the Wide Trail strategy [9].

The design is inspired by the Whirlpool hash function [2]. With this new design, we want to provide a higher security, keeping the performance at the same level. Since the selection of AES, there have been several block cipher [1], stream cipher [7] and hash function proposals [3,6,12] based on its design principles. In this design we have incorporated the feedback about the security level and implementation issues of AES-based designs. This has led to some innovations which will be detailed in the following section.

1.1 Motivation

A natural question to ask is whether it makes sense to propose a new hash function design one year after the start of the SHA-3 competition organized by NIST in order to come up with a new hash function standard. We have been following the competition actively and we have seen that many of the submissions have been broken already. The methods to cryptanalyse hash functions have been improved very significantly since the start of the competition. Hence it is not unthinkable that after the competition all submissions are either broken, very slow or not acceptable because of other reasons.

The Whirlwind design takes into account the recent development in hash function cryptanalysis, in particular the rebound attack, and adds a security margin as a precaution against possible further improvements. We employ large S-boxes (16-bit inputs and outputs) for low-probability differential trails. An implementation can still be very efficient due to our special choice of basis for the finite field GF(2^{16}) which speeds up implementations that compute the S-box entries instead of storing them in a large lookup table. The formulas that we derive in this paper may also be useful to derive compact hardware implementations of AES and AES-based hash functions.

The diffusion map has the same optimal diffusion properties as the Whirlpool diffusion, but is chosen in a way that makes the algebraic description of the round transformation less simple. This should alleviate the concerns about the applicability of *algebraic attacks*. Furthermore, we show in this paper a potential weakness of diffusion maps based on circulant matrices.

1.2 Overview of the paper

We fully specify Whirlwind in Sect. 2. Next, in Sect. 3, we explain our construction for the S-box and describe how it can be implemented without large lookup tables. We discuss the security of Whirlwind in Sect. 4. Section 5 deals with practical implementation issues and gives some performance figures for software implementations and estimates of memory requirements on 8-bit platforms.

2 Specification of Whirlwind

2.1 Internal state

Whirlwind has an internal state of 1024 bits, which can be represented by an 8×8 array of 16-bit elements:

$$a = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \end{bmatrix} \equiv \left[a_{i,j}\right]_{i,j=0}^{7}.$$

Each of the 16-bit elements of the state can in turn be represented by a 2×2 array of 4-bit elements:

$$\forall i, j\colon a_{i,j} = \begin{bmatrix} a_{i,j,0,0} & a_{i,j,0,1} \\ a_{i,j,1,0} & a_{i,j,1,1} \end{bmatrix}.$$

The i-th row of the state is denoted by a_i or by $a_{i,*,*,*}$.

2.1.1 Representation of finite fields

Elements of GF(2^4) are expressed in terms of a tower field decomposition to GF(2^2) and GF(2) using normal bases at each level. Let GF(2^2) be described by adjoining a root w of the primitive polynomial $X^2 + X + 1$ over GF(2) and analogously GF(2^4) by the adjunction of a root x of the primitive polynomial $X^4 + X + 1$ over GF(2). All nonzero elements of GF(2^2) and GF(2^4) can then be expressed in terms of powers of w and x, respectively. The elements $\alpha := w \in \mathrm{GF}(2^2)$ and $\beta := x^7 \in \mathrm{GF}(2^4)$ are normal, i.e. their conjugates form a normal basis over the respective subfield. The normal bases for the tower field decomposition are then given by $\{\alpha, \alpha^2\}$ for the construction of GF(2^2) over GF(2) and $\{\beta, \beta^4\}$ for the construction of GF(2^4) over GF(2^2). This choice of bases is further detailed and put into a general perspective in Sect. 3.

An element of GF(2^4) $\cong$ GF$(2^2)^2$ $\cong$ $\left(\mathrm{GF}(2)^2\right)^2$ in this normal bases representation is identified with the integer corresponding to the sequence of its coefficients in GF(2) and is written in hexadecimal. For example, 6_x corresponds to $((0, 1), (1, 0))$ which is $\left(0 + 1 \cdot \alpha^2, 1 \cdot \alpha + 0\right) = (w^2, w)$ in GF$(2^2)^2$ and $w^2 \cdot \beta + w \cdot \beta^4 = x^6$ in GF(2^4).

2.2 The round transformation

The round transformation consists of four maps, which are described below.

2.2.1 The nonlinear substitution layer γ

The nonlinear layer acts in parallel on the 64 elements of 16 bits. Each of these elements is interpreted as an element of GF(2^{16}) and replaced by its multiplicative inverse:

$$\gamma\colon \mathrm{GF}(2^{16})^{8\times 8} \to \mathrm{GF}(2^{16})^{8\times 8}\colon \left[a_{i,j}\right]_{i,j=0}^{7} \mapsto \left[(a_{i,j})^{-1}\right]_{i,j=0}^{7},$$

with the additional convention that zero is mapped to zero.

2.2.2 The linear maps θ and τ

The linear map τ acts as a transposition on the 8×8 matrix that represents the state:

$$\tau\colon \mathrm{GF}(2^{16})^{8\times 8} \to \mathrm{GF}(2^{16})^{8\times 8}\colon \left[a_{i,j}\right]_{i,j=0}^{7} \mapsto \left[a_{j,i}\right]_{i,j=0}^{7}.$$

Note that the explicit implementation of τ can be avoided by implementing two different round transformations: one where θ acts on the rows, and one where θ acts on the columns.

The linear map θ acts in parallel on the 8 rows of the state:

$$\theta\colon \mathrm{GF}(2^{16})^{8\times 8} \to \mathrm{GF}(2^{16})^{8\times 8}\colon \left[a_{i,j}\right]_{i,j=0}^{7} \mapsto \left[\lambda(a_i)|_j\right]_{i,j=0}^{7}.$$

We can also write this as follows:

$$\theta(a) = b \Leftrightarrow \forall i\colon \lambda(a_i) = b_i.$$

Furthermore, λ acts in parallel on the 4-bit subcomponents of the $a_{i,j}$:

$$\lambda(a_i) = b_i \Leftrightarrow \begin{cases} \lambda_0(a_{i,*,0,0}) = b_{i,*,0,0} \\ \lambda_1(a_{i,*,0,1}) = b_{i,*,0,1} \\ \lambda_1(a_{i,*,1,0}) = b_{i,*,1,0} \\ \lambda_0(a_{i,*,1,1}) = b_{i,*,1,1} \end{cases}$$

The maps λ_0, λ_1 are defined as follows:

$$\lambda_0\colon \mathrm{GF}(2^4)^{1\times 8} \to \mathrm{GF}(2^4)^{1\times 8}\colon a_{i,*,k,k} \mapsto a_{i,*,k,k} \cdot M_0$$
$$\lambda_1\colon \mathrm{GF}(2^4)^{1\times 8} \to \mathrm{GF}(2^4)^{1\times 8}\colon a_{i,*,k,1-k} \mapsto a_{i,*,k,k} \cdot M_1$$

with

$$M_0 = \text{dyadic}\,(5_x, 4_x, \mathtt{A}_x, 6_x, 2_x, \mathtt{D}_x, 8_x, 3_x) \text{ and}$$
$$M_1 = \text{dyadic}\,(5_x, \mathtt{E}_x, 4_x, 7_x, 1_x, 3_x, \mathtt{F}_x, 8_x),$$

where dyadic (s) denotes the dyadic matrix S corresponding to the sequence s over $\mathrm{GF}(2^4)$, i.e. $S_{i,j} = s_{i \oplus j}$. For example, writing out M_0 in full gives:

$$M_0 = \begin{bmatrix} 5_x & 4_x & \mathtt{A}_x & 6_x & 2_x & \mathtt{D}_x & 8_x & 3_x \\ 4_x & 5_x & 6_x & \mathtt{A}_x & \mathtt{D}_x & 2_x & 3_x & 8_x \\ \mathtt{A}_x & 6_x & 5_x & 4_x & 8_x & 3_x & 2_x & \mathtt{D}_x \\ 6_x & \mathtt{A}_x & 4_x & 5_x & 3_x & 8_x & \mathtt{D}_x & 2_x \\ 2_x & \mathtt{D}_x & 8_x & 3_x & 5_x & 4_x & \mathtt{A}_x & 6_x \\ \mathtt{D}_x & 2_x & 3_x & 8_x & 4_x & 5_x & 6_x & \mathtt{A}_x \\ 8_x & 3_x & 2_x & \mathtt{D}_x & \mathtt{A}_x & 6_x & 5_x & 4_x \\ 3_x & 8_x & \mathtt{D}_x & 2_x & 6_x & \mathtt{A}_x & 4_x & 5_x \end{bmatrix}.$$

Note that this approach for constructing a linear diffusion layer is novel. The functions λ_0, λ_1 act on elements of $\mathrm{GF}(2^4)$ and have a very simple and elegant description. The functions λ, θ inherit the optimal diffusion properties of the λ_i maps. However, if they are described as acting on elements of $\mathrm{GF}(2^{16})$, like the other components of the round transformation, then this requires the use of a linearized polynomial, rather than a simple matrix multiplication. This feature should alleviate concerns about the abuse of simple descriptions to mount *algebraic attacks*.

2.2.3 The affine layer σ^r

The affine layer adds a round-dependent constant c^r to the state, in order to break the symmetry between different positions in the state.

$$\sigma^r\colon \mathrm{GF}(2^{16})^{8\times 8} \to \mathrm{GF}(2^{16})^{8\times 8}\colon \left[a_{i,j}\right]_{i,j=0}^{7} \mapsto \left[a_{i,j} + c_{i,j}^r\right]_{i,j=0}^{7},$$

with $c^r = \gamma(s^r)$ and

$$\begin{aligned} s_{0,j}^r &= 8(r-1) + j,\ 0 \le j \le 7,\\ s_{i,j}^r &= 0, \qquad\qquad\quad 1 \le i \le 7, 0 \le j \le 7. \end{aligned}$$

Here, the integer value defining $s_{0,j}^r$ corresponds to the element of $\mathrm{GF}(2^{16})$ obtained by interpreting the sequence of its binary representation according to the normal bases decomposition of $\mathrm{GF}(2^{16})$ described in Sect. 3.6.

2.3 The compression function

The compression function φ takes as input a 512-bit chaining variable h and a 512-bit message block m. It outputs the updated chaining variable g. Both the chaining variable and the message blocks are represented by 8×4 arrays of 16-bit elements.

$$\varphi\colon \mathrm{GF}(2^{16})^{8\times 4} \times \mathrm{GF}(2^{16})^{8\times 4} \to \mathrm{GF}(2^{16})^{8\times 4}\colon (h, m) \mapsto g = \varphi(h, m).$$

The updated chaining variable is computed as follows.

1. Initialize the state a:
$$\begin{cases} a_{i,j} = h_{i,j} \\ a_{i+4,j} = m_{i,j} \end{cases}, \quad 0 \le i < 4,\ 0 \le j < 8.$$
2. Apply 12 iterations of the round transformation, which consists of the sequence $\gamma, \theta, \tau, \sigma^r$:
$$b = \left(\bigcirc_1^{r=12} \left(\sigma^r \circ \tau \circ \theta \circ \gamma\right)\right)(a).$$
Here, the notation $\bigcirc_m^{r=n} f_r$, with $m \le n$, denotes $f_n \circ f_{n-1} \circ \cdots \circ f_{m+1} \circ f_m$ for a sequence of functions $f_m, f_{m+1}, \ldots, f_{n-1}, f_n$.
3. Truncate and add the feed-forward:
$$g_{i,j} = b_{i,j} + h_{i,j},\ 0 \le i < 4,\ 0 \le j < 8.$$

Figure 1 illustrates the compression function.

2.4 Output transformation

An output transformation is required in order to avoid trivial length extension attacks. The output transformation of Whirlwind consists of an extra application of φ, with the message block equal to the 8×4 null matrix.

2.5 Truncation encoded in the initialization vector

Whirlwind outputs a digest of 512 bits. Some applications explicitly truncate digests to a smaller size. Other applications take the digest as an encoding of an integer number, and use

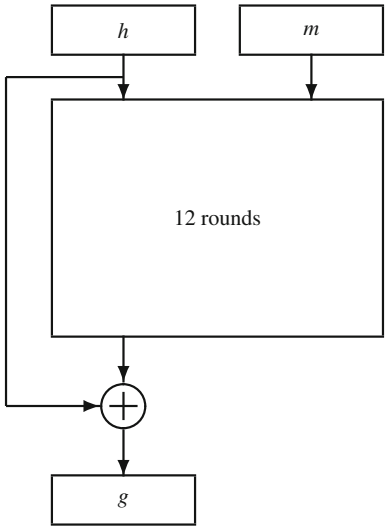

Fig. 1 $\varphi(h, m)$, the compression function of Whirlwind

this number in an operation modulo N. This corresponds to an implicit 'truncation' of the digest to a value between 0 and $N - 1$.

In order to avoid certain attacks based on explicit or implicit truncation [30], Whirlwind uses a mechanism to derive adaptable initialization vectors (IVs). The derivation works as follows. Assume that the output of Whirlwind will be truncated to $\log_2(N)$ bits, or reduced to an integer value between 0 and $N - 1$. Then N is converted to an 8×4 array of 16-bit elements and the initialization vector h^0 is computed as follows:

$$h^0 = \varphi(0, N).$$

An application can compute h^0 at the start of the digest computation, or it can store the initialization vectors for the N-values that it expects to meet most frequently.

2.6 Computing a digest

The full algorithm to compute a message digest is as follows.

Input: message m of bit length $L < \sqrt{N}$, L, truncation/reduction value N
Algorithm:

1. Pad m by adding a 1-bit, then as few 0-bits as necessary to obtain a bit string whose length is an odd multiple of 256, and finally with the 256-bit right-justified binary representation of L.
2. Split the padded message in $t = \lceil (L + 257)/512 \rceil$ blocks of 512 bits, denoted by m^i, $i = 0, \ldots, t - 1$.
3. Compute the Initialization Vector:
   $h^0 = \varphi(0, N)$.
4. Process the t message blocks:
   $h^{i+1} = \varphi(h^i, m^i),\ i = 0, \ldots, t - 1$.
5. Apply the output transformation
   $h^{t+1} = \varphi(h^t, 0)$.
6. Output $h^{t+1} \bmod N$.

Table 1 Test vectors for Whirlwind

Digest size in bits	Hash value of the empty string ($L = 0$)
256	`23b61b32a3b2abe0475e6e7585dd300d`
	`3643f9c36da9c22e42dace50d01d0012`
512	`4dfe5a37c81711cdb9abe7aaffd81727`
	`215801642b88eca606214277d1b3ba11`
	`5220d074d153965e692e991326e508af`
	`365cb9aaba97b36b2116c156012b1397`

2.6.1 Test vectors

Input–output pairs for the common digest sizes of 256 and 512 bits are provided in Table 1.

3 Efficient implementations by using normal basis arithmetic

Here we briefly summarize some properties of normal bases over finite fields. The finite field $\mathrm{GF}(2^{mp})$ is isomorphic to a p-dimensional vector space over $\mathrm{GF}(2^m)$. This implies that it is possible to construct a *basis* for $\mathrm{GF}(2^{mp})$. A basis consists of p elements $\beta_0, \beta_1, \ldots, \beta_{p-1} \in \mathrm{GF}(2^{mp})$ such that all elements of $\mathrm{GF}(2^{mp})$ can be written as a linear combination of the elements β_j, with all coefficients elements of $\mathrm{GF}(2^m)$. As in all vector spaces, there are many different choices possible for the basis, and the choice of basis may influences the complexity to describe transformations on the vector space.

3.1 Normal bases

A *normal basis* is constructed by choosing an element $v \in \mathrm{GF}(2^{mp})$ and setting $v^{2^{mj}}, \quad 0 \leq j \leq p-1$. Not all elements of $\mathrm{GF}(2^{mp})$ result in a basis over $\mathrm{GF}(2^m)$, but there exist always some suitable elements. Let now

$$x = \sum_{j=0}^{p-1} c_j v^{2^{mj}}, \quad c_j \in \mathrm{GF}(2^m).$$

We raise both sides to the power 2^m. This operation is linear over $\mathrm{GF}(2^{mp})$ and corresponds to the identity transformation for all elements in $\mathrm{GF}(2^m)$. Then we obtain

$$x^{2^m} = \sum_{j=0}^{p-1} (c_j)^{2^m} (v^{2^{mj}})^{2^m} = \sum_{j=0}^{p-1} c_j v^{2^{m(j+1)}} = \sum_{j=0}^{p-1} c_{j-1} v^{2^{mj}}.$$

In words, this corresponds to the following property.

Property 1 ([19]) *If the elements of the finite field* $\mathrm{GF}(2^{mp})$ *are represented by* p*-dimensional vectors over* $\mathrm{GF}(2^m)$ *using a normal basis, then raising an element to the power* 2^m *corresponds to rotating the coordinates of the element by one position.*

Let now $m = 1$ and consider any power map $power: x \to x^n$. For example the inversion map used in the AES S-box i.e. $s(x) = x^{-1}$ in $\mathrm{GF}(2^8)$ or equivalently $s(x) = x^{254}$. Clearly, $power(x^2) = (power(x))^2$. Hence we obtain the following property.

Property 2 ([24]) *If the elements of the finite field* GF(2^{mp}) *are represented in a normal basis, then any power map* $power(x)$ *is rotation invariant:*

$$rot(power(x)) = power(rot(x)).$$

We use another property of normal bases.

Property 3 ([27]) *If* v *is a normal element of* GF(q^{mp}) *with respect to (w.r.t.)* GF(q), *then* $w = Tr_{\mathrm{GF}(q^{mp})/\mathrm{GF}(q^m)}(v)$ *is a normal element of* GF(q^m) *w.r.t.* GF(q).

Proof Recall that $w = Tr_{\mathrm{GF}(q^{mp})/\mathrm{GF}(q^m)}(v) = \sum_{i=0}^{p-1} v^{q^{mi}}$. We will show that the conjugates w^{q^j} $(j = 0, \ldots, p-1)$ are linearly independent. Indeed

$$w^{q^j} = \left(\sum_{i=0}^{p-1} v^{q^{mi}}\right)^{q^j} = \sum_{i=0}^{p-1} (v^{q^{mi}})^{q^j} = \sum_{i=0}^{p-1} v^{q^{mi+j}},$$

so each of the conjugates of w is a sum of p different conjugates of v, and all the mp conjugates of v appear exactly once in the m sums of p summands each. Since v was chosen to be a normal element of GF(q^{mp}) over GF(q), the conjugates of v are linearly independent over GF(q). Consequently, w is a normal element of GF(q^m) w.r.t. GF(q). □

3.2 Normal bases in Whirlwind

For the rest of this section, we set $p = 4$. We need two elements v_2 and v_4 as follows. Let v_2 be an element of GF(2^{2m}), such that

- $\{v_2, v_2^{\ell_1}\}$ with $\ell_1 = 2^m$ be a normal basis of GF(2^{2m}) over GF(2^m). Define:
- $q_2 = Tr_{\mathrm{GF}(2^{2m})/\mathrm{GF}(2^m)}(v_2) = v_2 + v_2^{\ell_1}$, so $q_2 (\neq 0) \in \mathrm{GF}(2^m)$ and
- $g_2 = q_2^{-1} v_2^2 + v_2 = q_2^{-1} v_2^{\ell_1+1}$, i.e. $g_2 (\neq 0) \in \mathrm{GF}(2^m)$.

Let v_4 be an element of GF(2^{4m}) such that

- $\{v_4, v_4^{\ell_2}\}$ with $\ell_2 = 2^{2m}$ be a normal basis of GF(2^{4m}) over GF(2^{2m}). Define:
- $q_4 = Tr_{\mathrm{GF}(2^{4m})/\mathrm{GF}(2^{2m})}(v_4) = v_4 + v_4^{\ell_2}$, so $q_4 (\neq 0) \in \mathrm{GF}(2^{2m})$ and
- $g_4 = q_4^{-1} v_4^2 + v_4 = q_4^{-1} v_4^{\ell_2+1}$, i.e. $g_4 \in \mathrm{GF}(2^{2m})$.

The elements v_2 and v_4 can be chosen independently or we can choose a normal element $v_4 \in \mathrm{GF}(2^{4m})$ and then derive $v_2 = Tr_{\mathrm{GF}(2^{4m})/\mathrm{GF}(2^{2m})}(v) \in \mathrm{GF}(2^{2m})$. Using Property 3 it follows that v_2 is a normal element in GF(2^{2m}). Note that in this case we have $q_4 = v_2$ and $q_2 = Tr_{\mathrm{GF}(2^{2m})/\mathrm{GF}(2^m)}(v_2) = Tr_{\mathrm{GF}(2^{4m})/\mathrm{GF}(2^m)}(v)$ (using trace transitivity) and hence $q_4, q_2 \neq 1$ (Property 3 implies that q_2 is a normal element in GF(2^m)).

We show now (analogously to [24]) that the use of a normal basis leads to simple formulas for products and inverses of elements.

3.3 Multiplication

Let (a, b) and (c, d) be the coordinates of two elements of GF(2^{2m}). Therefore coordinates of the product are given by the following formula:

$$\begin{aligned}
(e, f) = (a, b) \times (c, d) &\Leftrightarrow ev_2 + fv_2^{\ell_1} = acv_2^2 + (ad + bc)v_2^{\ell_1+1} + bdv_2^{2\ell_1} \\
&\Leftrightarrow ev_2 + fv_2^{\ell_1} = q_2acv_2 + (ac + ad + bc + bd)v_2^{\ell_1+1} + q_2bdv_2^{\ell_1} \\
&\Leftrightarrow ev_2 + fv_2^{\ell_1} = q_2acv_2 + (a + b)(c + d)v_2^{\ell_1+1} + q_2bdv_2^{\ell_1} \\
&\Leftrightarrow ev_2 + fv_2^{\ell_1} = q_2acv_2 + (a + b)(c + d)(g_2v_2 + g_2v_2^{\ell_1}) \\
&\qquad\qquad + q_2bdv_2^{\ell_1} \\
&\Leftrightarrow \begin{cases} e = (a + b)(c + d)g_2 + q_2ac \\ f = (a + b)(c + d)g_2 + q_2bd \end{cases}
\end{aligned}$$

Let now (a, b) and (c, d) be the coordinates of two elements of GF(2^{4m}), the product in this case is given by the following formula:

$$(e, f) = (a, b) \times (c, d) \Leftrightarrow \begin{cases} e = (a + b)(c + d)g_4 + q_4ac \\ f = (a + b)(c + d)g_4 + q_4bd \end{cases}$$

3.4 Inversion

Let (a, b) be the coordinates of an element of GF(2^{2m}). The coordinates of the inverse element are given by the following formula:

$$\begin{aligned}
(c, d) = (a, b)^{-1} &\Leftrightarrow 1 = (av_2 + bv_2^{\ell_1})(cv_2 + dv_2^{\ell_1}) \\
&\Leftrightarrow q_2^{-1}v_2 + q_2^{-1}v_2^{\ell_1} = ((a + b)(c + d)g_2 + q_2ac)v_2 \\
&\qquad\qquad + ((a + b)(c + d)g_2 + q_2bd)v_2^{\ell_1} \\
&\Leftrightarrow \begin{cases} q_2^{-1} = (a + b)(c + d)g_2 + q_2ac \\ q_2^{-1} = (a + b)(c + d)g_2 + q_2bd \end{cases} \\
&\Leftrightarrow \begin{cases} 0 = q_2(ac + bd) \\ 1 = (a + b)(c + d)g_2q_2 + q_2^2bd \end{cases} \\
&\Leftrightarrow \begin{cases} c = ba^{-1}d \\ 1 = (a + b)(c + d)g_2q_2 + q_2^2bd \end{cases} \\
&\Leftrightarrow \begin{cases} c = ba^{-1}d \\ 1 = (a + b)(ba^{-1} + 1)g_2q_2d + q_2^2bd \end{cases} \\
&\Leftrightarrow \begin{cases} c = ba^{-1}d \\ a = (a + b)(b + a)g_2q_2d + q_2^2abd \end{cases} \\
&\Leftrightarrow \begin{cases} c = ba^{-1}d \\ d = ((a + b)^2g_2q_2 + q_2^2ab)^{-1}a \end{cases} \\
&\Leftrightarrow \begin{cases} c = ((a + b)^2g_2q_2 + q_2^2ab)^{-1}b \\ d = ((a + b)^2g_2q_2 + q_2^2ab)^{-1}a \end{cases}
\end{aligned}$$

Analogously let (a, b) be the coordinates of an element of GF(2^{4m}). The coordinates of the inverse element in this case are given by the following formula:

$$(c, d) = (a, b)^{-1} \Leftrightarrow \begin{cases} c = ((a+b)^2 g_4 q_4 + q_4^2 ab)^{-1} b \\ d = ((a+b)^2 g_4 q_4 + q_4^2 ab)^{-1} a \end{cases} \quad (1)$$

In this way we can decompose the inversion map from GF(2^{4m}) $\rightarrow$ GF(2^{2m}) $\rightarrow$ GF(2^m). For the Whirlwind proposal $m = 4$ so we get the decomposition GF(2^{16}) $\rightarrow$ GF(2^8) $\rightarrow$ GF(2^4).

3.5 Inversion and multiplication in GF(16)

A normal basis is called *optimal normal basis* (*ONB*) [22], when the complexity of the multiplication formula in this basis is minimal, i.e. equal to $2n - 1 = 7$ [15]. In the non-optimal normal basis (NB), the complexity of the multiplication formula equals 9 [26]. In these two cases multiplying $x = (x_0, x_1, x_2, x_3)$ with $y = (y_0, y_1, y_2, y_3)$ results in $z = (z_0, z_1, z_2, z_3)$, where

$$\begin{aligned} &\text{NB:} \quad z_3 = x_2y_3 + x_3y_2 + x_1y_3 + x_3y_1 + x_3y_0 + x_0y_3 + x_2y_2 + x_0y_1 + x_1y_0. \\ &\text{ONB:} \quad z_3 = x_3y_1 + x_0y_1 + x_0y_2 + x_1y_3 + x_1y_0 + x_2y_0 + x_2y_2. \end{aligned} \quad (2)$$

As noted by Paar [26] the multiplication in any normal basis is rotation symmetric, so the rest of the output bits z_0, z_1 and z_2 can be computed by rotating the input bits.

Table 2 gives the inverses in both cases (in normal and optimal normal basis). One way to obtain the values in the table is to use the formulas derived in [24]. Denote $(f_3, f_2, f_1, f_0) = \mathrm{s}(x_3, x_2, x_1, x_0)$, due to the rotational symmetry of s, the other output bits f_3, f_2 and f_1 can be computed by rotating the input bits. The Algebraic Normal Form (ANF) of each output bit f_0 is given by:

$$\begin{aligned} &\text{NB:} \quad f_0 = x_0 + x_3 + x_0x_1 + x_1x_3 + x_0x_1x_2 + x_0x_1x_3 + x_1x_2x_3 \\ &\text{ONB:} \quad f_0 = x_1 + x_0x_3 + x_0x_2 + x_1x_3 + x_0x_1x_2 + x_0x_1x_3 + x_1x_2x_3 \end{aligned} \quad (3)$$

Table 2 The inversion $\mathrm{s}(x)$ in normal basis and in optimal normal basis

x	NB	ONB
0000	0000	0000
0001	0011	0100
0010	0110	1000
0011	0001	1110
0100	1100	0001
0101	1010	1010
0110	0010	1101
0111	1101	1001
1000	1001	0010
1001	1000	0111
1010	0101	0101
1011	1110	1100
1100	0100	1011
1101	0111	0110
1110	1011	0011
1111	1111	1111

Table 3 Overview of the normal bases decomposition in Whirlwind

(a) Field representations used for unique reference

Field	Defining polynomial
$\mathrm{GF}(2^{16}) \cong \mathrm{GF}(2)(z)$	$X^{16} + X^5 + X^3 + X^2 + 1$
$\mathrm{GF}(2^8) \cong \mathrm{GF}(2)(y)$	$X^8 + X^4 + X^3 + X^2 + 1$
$\mathrm{GF}(2^4) \cong \mathrm{GF}(2)(x)$	$X^4 + X + 1$
$\mathrm{GF}(2^2) \cong \mathrm{GF}(2)(w)$	$X^2 + X + 1$

(b) Normal basis decomposition

Field	Normal element	q_i	g_i
$\mathrm{GF}(2^{16})$	$v_4 = z^{101}$	–	–
Basis over $\mathrm{GF}(2^8)$: $\{v_4, v_4^{256}\}$			
$\mathrm{GF}(2^8)$	$v_2 = y^{101}$	$q_4 = v_2$	$g_4 = 1$
Basis over $\mathrm{GF}(2^4)$: $\{v_2, v_2^{16}\}$			
$\mathrm{GF}(2^4)$	$v_1 = x^7$	$q_2 = v_1$	$g_2 = x^4$
Basis over $\mathrm{GF}(2^2)$: $\{v_1, v_1^4\}$			
$\mathrm{GF}(2^2)$	$v_0 = w$	$q_1 = v_0$	$g_1 = 1$
Basis over $\mathrm{GF}(2)$: $\{v_0, v_0^2\}$			
$\mathrm{GF}(2)$			

3.6 Basis choice for Whirlwind

In Whirlwind, $\mathrm{GF}(2^{16})$ is recursively decomposed into smaller subfields according to Sect. 3.2 by choosing a normal element of $\mathrm{GF}(2^{16})$ and using the traces of this element into the subfields to construct the normal bases. This decomposition is employed uniformly at each level, so that all individual field extensions have degree two:

$$\mathrm{GF}(2^{16}) \to \mathrm{GF}(2^8) \to \mathrm{GF}(2^4) \to \mathrm{GF}(2^2) \to \mathrm{GF}(2).$$

In order to describe our choice for the normal bases unambiguously, we use the field representations given in Table 3a as a reference. The normal bases used in Whirlwind, together with the elements q_i, g_i used for the field arithmetic are summarised in Table 3b.

4 Security analysis

Designing a provably secure hash function is still beyond the state of the art. Reductionist security arguments provide provable guarantees, but do not exclude attacks outside the model, the algorithm VSH being a prominent example [8,29].

Recently some reduced-round versions of Whirlpool were cryptanalyzed [18,21]. In this section, we argue that Whirlwind is secure against the currently known cryptanalytic attacks by applying the most important state-of-the-art methods of cryptanalysis and investigating their complexity.

4.1 Resistance against basic differential attacks

The resistance of hash functions against differential attacks is typically analyzed by studying the *expected differential probability* (EDP) of the differential trails through the compression function. There are a number of theoretical problems associated with using the EDP to say something about hash functions. However, to date the EDP of differential trails is the only measure known that can be used to assess the security of a primitive against differential cryptanalysis and that can be computed efficiently. In this section, we bound the EDP of differential trails through the compression function of Whirlwind and we explain what meaning can be attributed to this number.

4.1.1 Terminology

Let $B(x)$ denote a function over $\mathrm{GF}(2^n)$ composed of r steps $f^i(x)$:

$$B(x) = \left(f^r \circ \cdots \circ f^1\right)(x).$$

A *differential trail* through $B(x)$ is a vector $Q = (b^0, b^1, \ldots, b^r)$ such that

$$\begin{aligned} f^1(x + b^0) &= f^1(x) + b^1 \\ &\vdots \\ \left(f^r \circ \cdots \circ f^1\right)(x + b^0) &= \left(f^r \circ \cdots \circ f^1\right)(x) + b^r. \end{aligned} \tag{4}$$

The differential probability $\mathrm{DP}(Q)$ of a characteristic Q with respect to $B(x)$ is defined as

$$\mathrm{DP}(Q) = 2^{-n} \# \left\{x \in \mathrm{GF}(2^n) \mid x \text{ satisfies (4)}\right\}.$$

Assume that the functions f^i are parameterized by keys k^i and denote by $\mathcal{K}$ the space of the vectors $k = (k^1, k^2, \ldots, k^r)$. This leads in a straightforward way to the definition of a parameterized probability $\mathrm{DP}[k](Q)$ of a differential trail. The *expected differential probability* (*EDP*) of a differential trail Q is defined as the mean value of $\mathrm{DP}[k](Q)$:

$$\mathrm{EDP}(Q) = \mathbb{E}(\mathrm{DP}[k](Q); k) = 2^{-|\mathcal{K}|} \sum_{k \in \mathcal{K}} \mathrm{DP}[k](Q).$$

Here, k is assumed to be a uniformly distributed random variable taking values in $\mathcal{K}$. In block cipher design papers, it is usually assumed that the variance of $\mathrm{DP}[k](Q)$ is small, such that

$$\mathrm{DP}[k](Q) \approx \mathrm{EDP}(Q), \quad \forall k. \tag{5}$$

4.1.2 Implications for Whirlwind

The compression function of a hash function like Whirlwind (barring the feed-forward) can be considered as a block cipher with all the round keys set to zero. Hence, by (5), the fraction of pairs that follow a Q is given by

$$\mathrm{DP}[k = (0, \ldots, 0)](Q) \approx \mathrm{EDP}(Q).$$

The design of Whirlwind follows the Wide Trail design strategy [9]. The linear maps θ and τ ensure that a differential trail over R rounds contains at least $\lfloor R/4 \rfloor \mathcal{B}^2$ active S-boxes, where $\mathcal{B}$ is the differential branch number of θ, i.e. 9. The *expected differential probability*

(EDP) of a differential trail Q through 4 rounds of Whirlwind can be upper bounded as follows:

$$\mathrm{EDP}(Q) \leq \left(2^{-14}\right)^{8^2} = 2^{-1134}.$$

If the fact that only a very small fraction of pairs follows a given differential trail implies that it is difficult to find such a pair, then the bound given here implies that it is very difficult to find such pairs for Whirlwind.

4.2 Why θ uses dyadic matrices instead of circulant matrices

Since the designs of Square and Rijndael [9] have been published, the use of circulant matrices in diffusion layers has become widespread, with some exceptions though [1]. We explain here why we prefer dyadic matrices.

Although (5) is often used as a starting assumption, it has never been demonstrated that there exist block ciphers for which it holds. On the contrary, several counterexamples have been found. For instance, in [10] it was shown that for AES reduced to four rounds, there is a significant fraction of differential trails, *plateau trails*, for which the distribution of $\mathrm{DP}[k](Q)$ is bimodal, and has a large variance. The larger the variance of $\mathrm{DP}[k](Q)$ is, the higher the risk that $\mathrm{DP}[0](Q)$ will differ significantly from $\mathrm{EDP}(Q)$, and hence the less useful a bound on $\mathrm{EDP}(Q)$ becomes.

In [11] it was shown that the existence of plateau trails is due to the existence of *related differentials* in the linear diffusion layer. Let m be a linear map

$$m\colon (\mathrm{GF}(2^n))^{4t} \to (\mathrm{GF}(2^n))^{4t}\colon x \mapsto m(x).$$

Let $a, b \in (\mathrm{GF}(2^n))^{4t}$. The differentials $(a, m(a)), (b, m(b)), (a + b, m(a) + m(b))$ are *related differentials* over m if and only if

$$\begin{aligned} a_j b_j (a_j + b_j) &= 0, \quad \forall j \\ m(a)|_j m(b)|_j \left(m(a)|_j + m(b)|_j\right) &= 0, \quad \forall j. \end{aligned}$$

We prove below that related differentials always exist in diffusion layers that consist of the multiplication by an 8×8 circulant matrix. That is the reason why we chose to use in Whirlwind a dyadic matrix instead.

Theorem 1 *A linear map*

$$m\colon (\mathrm{GF}(2^n))^{4q} \to (\mathrm{GF}(2^n))^{4q}\colon x \mapsto m(x) = xM$$

with M a circulant matrix of dimensions $4q \times 4q$, has related differentials.

Proof Let M be defined by $m_{i,j} = \alpha_{j-i \bmod 4q}$. Define a, b:

$$\begin{aligned} a_j &= \alpha_j \quad \text{if } j \text{ is even, else } a_j = 0 \\ b_j &= 0 \quad \text{if } j \text{ is even, else } b_j = \alpha_j. \end{aligned}$$

Then obviously $a_j b_j (a_j + b_j) = 0, \forall j$.

Secondly, denote $c = aM$ and $d = bM$. It follows that:

$$c_j = \sum_{t=0}^{4q-1} a_t m_{t,j} = \sum_{u=0}^{2q-1} \alpha_{2u \bmod 4q} \alpha_{j-2u \bmod 4q} \tag{6}$$

$$d_j = \sum_{t=0}^{4q-1} b_t m_{t,j} = \sum_{v=0}^{2q-1} \alpha_{2v+1 \bmod 4q} \alpha_{j-1-2v \bmod 4q}. \tag{7}$$

We now consider three different cases:

$j = 2i + 1$: We apply a change of variable in (7): $w = i + 2q - v$. Since

$$\begin{aligned} \alpha_{2v+1 \bmod 4q} &= \alpha_{2i+1-2w+4q \bmod 4q} = \alpha_{2i+1-2w \bmod 4q} \\ \alpha_{2i-2v \bmod 4q} &= \alpha_{2w-4q \bmod 4q} = \alpha_{2w \bmod 4q} \\ v = 0 &\Leftrightarrow w = i + 2q \\ v = 2q - 1 &\Leftrightarrow w = i + 1, \end{aligned}$$

we obtain:

$$d_j = \sum_{w=i+1}^{i+2q} \alpha_{2w \bmod 4q} \alpha_{j-2w \bmod 4q}$$

This sum contains the same terms as (6) but in a different order. Hence we have $c_j = d_j$ when j is odd.

$j = 4s + 2$: We rewrite (6) as follows:

$$c_{4s+2} = \sum_{f=0}^{q-1} \alpha_{4f \bmod 4q} \alpha_{4s+2-4f \bmod 4q} + \sum_{g=0}^{q-1} \alpha_{4g+2 \bmod 4q} \alpha_{4s-4g \bmod 4q}$$

In the second sum, we replace $s - g$ by a new variable h and obtain:

$$c_{4s+2} = \sum_{f} \alpha_{4f \bmod 4q} \alpha_{4(s-f)+2 \bmod 4q} + \sum_{h} \alpha_{4(s-h)+2 \bmod 4q} \alpha_{4h \bmod 4q} = 0$$

Both sums are equal and hence $c_j = 0$ when $j = 4s + 2$.

$j = 4s$: We rewrite (7) as follows:

$$d_{4s} = \sum_{f=0}^{q-1} \alpha_{4f+1 \bmod 4q} \alpha_{4(s-f)-1 \bmod 4q} + \sum_{g=0}^{q-1} \alpha_{4g+3 \bmod 4q} \alpha_{4(s-g)-3 \bmod 4q}$$

Both sums are equal, and hence $d_j = 0$ when $j = 4s$.

It follows that $c_j d_j (c_j + d_j) = 0, \forall j$. □

4.3 Rebound attacks

In this section, we analyze the applicability of the rebound attack to Whirlwind. The rebound attack [21] is a new cryptanalytic attack on hash functions where the attacker attempts to fulfill the low-probability part of a truncated collision differential by exploiting available freedom in choosing concrete differences and values in this part first and then propagating those outwards through the parts with higher probability.

The rebound attack particularly lends itself to analyzing algorithms designed according to the wide trail design strategy where parts with very low probability cannot be avoided in differential trails at a certain length or above. In a rebound attack, such a part, where the full state is active, is then chosen to be in the middle of a multi-round truncated differential trail. Concrete differences and (pairs of) values propagating through this fully active state are efficiently found by matching inputs and outputs of the nonlinear layer for all S-boxes (the *match-in-the-middle* step). For the propagation of the obtained differences and values to the beginning and the end of the trail, it then has to be ensured that the patterns of active S-boxes follow the specified truncated differential. This implies that the probability of the outbound phase is completely determined by the propagation characteristics of the linear diffusion layer. For more details on the rebound attack, we refer to the original paper [21].

Before applying the rebound attack to Whirlwind, we discuss some preliminaries.

Finding a match for γ. The match-in-the-middle step of the inbound phase is commonly making use of a precomputed table to determine values actually following the chosen differential through the S-boxes. For each a and b, this table contains the solutions to the equation $s(x) \oplus s(x \oplus a) = b$. In case of the inversion mapping, either two or four solutions exist for each possible differential (a, b) and the probability that a particular differential exists is about $1/2$.

For Whirlwind's 16-bit S-box, the size of this table is 2^{37} bits, which is not always practical. However, the storage requirement can be lowered without significant increase in computational cost by using a lookup table of $2^{16} \times 2^{16}$ bits just specifying whether a particular differential exists or not. Once a differential match for all S-boxes has been found, only the solutions for the up to 64 different S-box differentials that are actually present have to be calculated. Doing this by brute force takes 2^{22} time, which is negligible since this immediately yields 2^{64} combinations of the two solutions found per individual S-box. In total, 2^{64} values of the state following the differential are expected to be found in $2^{64} + 2^{22}$ time, so the average cost for one match is still about 1.

In comparison with the designs using 8-bit inversion S-boxes, it is interesting to note that merely doubling the size of an individual S-box does not contribute significantly to the average cost of the match-in-the-middle step.

Propagation characteristics of θ. As stated before, the probability of the outbound phase essentially depends on the propagation of truncated differentials through θ. Since each λ_i is an MDS mapping, the sum of active and inputs and outputs is at least equal to 9. Any admissible transition of a active inputs and b active outputs (denoted $a \rightarrow b$) in fixed positions requiring b_0 of the b components to be zero occurs with a probability of about 2^{-4b_0}. Consequently, the probability of a $a \rightarrow b$ transition of θ with b_0 zero components in fixed positions is lower bounded by 2^{-16b_0}.

4.3.1 The basic differential trail

In what follows, we are using the description of the round function where the explicit calculation of τ is replaced by alternatingly applying θ to the rows (denoted θ_R) and the columns (denoted θ_C). Also note that in terms of differences, σ^r can be neglected and is therefore omitted from the sequence of operations.

In order to obtain a collision for the compression function of Whirlwind, we have to find a differential mapping differences only in the right half of the state (i.e. differences in the message) to a zero difference in the left half of the state (i.e. the output chaining value).

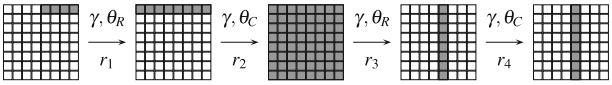

Fig. 2 Basic truncated differential trail covering 4 rounds

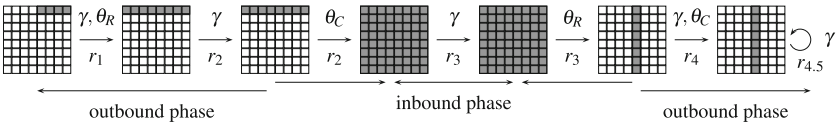

Fig. 3 The collision attack on 4.5 rounds

The basic trail of truncated differentials has the following pattern of active S-boxes which is also illustrated in Fig. 2:

$$4 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 8. \tag{8}$$

While this trail does not have the minimum number of active S-boxes according to the wide trail strategy, it minimizes the cost of the outbound phases of the attacks.

4.3.2 Semi-free-start collision on 4.5 rounds

The differential trail (8) can be directly used in a rebound attack to obtain a semi-free-start collision for Whirlwind reduced to 4.5 rounds. The attack (see also Fig. 3) goes as follows.

Inbound phase. The inbound phase of the attack covers the expensive fully active state in the middle of the trail. First, we choose a random difference for the 8 cells of the first row at the input of θ_C in the second round. By the MDS property, we obtain a fully active state at the beginning of round 3. Analogously, we choose another random difference of 8 cells for the fifth column at the output of θ_R in round 3 and propagate backwards to obtain another fully active state at the output of γ in round 3.

Using the procedure described above, now a match-in-the-middle is performed to obtain values at the input and output of the S-box layer in round 3 matching the differential. After trying about 2^{64} differences, we can expect to find one existing differential and 2^{64} conforming values.

Outbound phase. The differences and state values obtained in the previous step are now propagated outwards through the γ layers. From now on, we require those to follow certain truncated differential trails. In the backward direction, we require θ_R of round 1 to propagate the 8 active cells of the first row to 4 active cells in the right half of the row. The probability of this is lower bounded by 2^{-64}, since four out of eight cells in specific positions must have a zero difference. In the forward direction, θ_C needs to propagate a fully active column into a fully active column, which happens with a probability of at least $1 - \left(\sum_{i=1}^{7} 2^{-16i}\right)$, which is about 1. At the end, half a round (consisting of γ) can be appended for free since this does not change the activity pattern of the truncated differential.

Summarizing, we need to fulfill one $8 \rightarrow 4$ transition in the backward direction. The probability of the outbound phase is thus 2^{-64} so that the 2^{64} values provided by 2^{64} iterations of the inbound phase are just sufficient to get one pair following the trail. A semi-free-start collision for 4.5 rounds can hence be found with complexity 2^{64}.

4.3.3 Semi-free-start near-collision on 5.5 rounds

The semi-free-start collision attack on 4.5 rounds of Whirlwind can be extended to a semi-free-start near-collision attack on 5.5 rounds (see Fig. 4). Instead of the $8 \rightarrow 8$ transition in θ_C of round 4, we require a propagation of 8 to 1 active S-box in the first row. After θ_R in round 5, this expands to a fully active first row with probability 1 due to the MDS property. Since the output chaining value consists of the left half of the state, we obtain a near-collision on 448 of the 512 bits. The inbound phase and the backwards propagation part of the outbound phase are exactly the same as in the attack on 4.5 rounds, but the probability of the forwards propagation part decreases: It is now lower bounded by 2^{-112} since we require seven out of eight cells to have a zero difference.

In total, the outbound phase has a probability of 2^{-176} and the complexity of obtaining a near-collision pair for 5.5 rounds is 2^{176}, which is already quite close to a generic birthday attack on 448 bits.

4.3.4 Extension to 5.5/6.5 rounds

Both the collision and near-collision attacks previously described can be extended by prepending one round. In this round, θ is operating on the columns, so that the four active cells in the right half of the first row propagate to four fully active columns in the backwards part of the outbound phase with probability 1. The complexities of the previous attacks hence do not change when extended by one round at the beginning. Note however that such attacks only apply when starting at even round numbers, since we require a zero difference in the left half of the state, a constraint which is violated by having θ operate on rows in the first round. The near-collision attack on 6.5 rounds is illustrated in Fig. 5.

4.3.5 Extensions to more rounds

Rebound attacks on other wide-trail hash functions have basically used two strategies to extend attacks beyond the basic trail. Either a key or message schedule was used to afford more fully active states by exploiting the freedom available from there (e.g. Whirlpool [21]) or multiple independent inbound phases were efficiently connected by exploiting insufficient diffusion, for instance between parallel states (e.g. Lane [20]).

Both strategies seem inapplicable to Whirlwind due to the absence of influence on intermediate rounds via a key schedule and due to the fact that diffusion is performed according to the wide trail strategy on the whole state and not only on less interconnected parts of it.

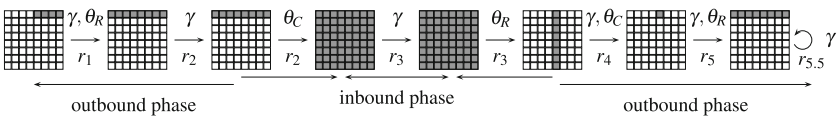

Fig. 4 The near-collision attack on 5.5 rounds

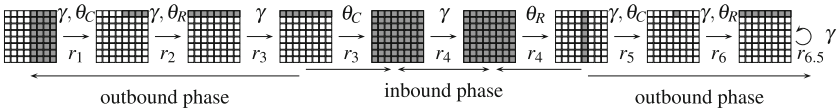

Fig. 5 The near-collision attack on 6.5 rounds

5 Performance figures and practical implementation considerations

The square shape of its state and the fact that it is designed according to the wide trail strategy imply that Whirlwind can be implemented using approaches developed for the AES [9] and especially Whirlpool [2]. We briefly recall the techniques here and discuss how to apply them to Whirlwind.

5.1 Software implementation on standard microprocessors

The standard approach to implement the linear layer θ is to use lookup tables containing all scalar products with each of the rows, where the S-box application is integrated on the scalars.

For Whirlwind, θ consists of the parallel application of four matrix multiplications over $\mathrm{GF}(2^4)^{8\times 8}$ on the rows of the state. Combining their contribution to one output row into a single table, both γ and θ can be implemented with eight table lookups. Denote by M_{ik} the k-th row of the dyadic matrices M_0 and M_1 and define the table T_k, $0 \le k \le 7$ by

$$T_k\left[\begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}\right] = s\left[\begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}\right] \cdot \left(\begin{pmatrix} M_{0k,0} & M_{1k,0} \\ M_{1k,0} & M_{0k,0} \end{pmatrix}, \ldots, \begin{pmatrix} M_{0k,7} & M_{1k,7} \\ M_{1k,7} & M_{0k,7} \end{pmatrix}\right). \quad (9)$$

Each row b_i of $b = (\theta \circ \gamma)(a)$ is then equal to

$$b_i = \bigoplus_{k=0}^{7} T_k[a_{i,k}]. \quad (10)$$

Due to the larger S-box, the memory requirements of each table is $2^{16} \cdot 128$ bits, i.e. one megabyte.

5.1.1 Impact of τ

In practice, each of the 2×2 submatrices comprising one element of the state will be identified by the vector $(x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1})$ and both a row of the state and the entries of the T_k will be organized in machine-sized words. For Whirlwind, each row then comprises two 64-bit or four 32-bit words. However, this layout of the state implies that operations across rows become significantly more efficient than operations across columns.

In designs using cyclic shifts to diffuse across second dimension, this operation can be implemented by a simple reordering of indices in (10), so that only row operations need to be performed. This is however not possible for a matrix transposition.

As noted in Sect. 2.2.2, the calculation of τ can be avoided by letting θ operate on rows and columns of the state alternatingly. The normal description of Whirlwind's round transformation:

$$b = (\sigma^r \circ \tau \circ \theta \circ \gamma)(a)$$

then turns into

$$b = \begin{cases} \left((\sigma^r)^T \circ \theta \circ \gamma\right)(a) & \text{for odd } r, \\ \left(\sigma^r \circ \theta_C \circ \gamma\right)(a) & \text{for even } r \end{cases}$$

where $(\sigma^r)^T$ denotes the application of the transpose of the r-th round constant and θ_C is θ operating on the columns.

In a state layout representing a row using machine words, the computation of θ_C still involves one matrix transposition, but the tables for θ can be reused and the number of transpositions is reduced to $R/2$, with R being the total number of rounds.

The transpose can for example be efficiently implemented using a recursive decomposition of the 8×8 transpose into transpositions of 4×4 and 2×2 matrices using SIMD instructions such as shuffle and unpack in Intel's SSE instruction set.

5.1.2 Performance of the table-based approach

While arguably being an efficient method to implement Whirlwind in theory, the memory required to store all eight lookup tables is 8 MB, which exceeds the L2 cache size available to a single core in most contemporary processors. As a result, parts of the table are constantly cached in and out, resulting in a significant performance penalty. As indicated in Table 4, the performance greatly varies with the size of the L2/L3 cache. A significant speedup is expected on CPUs with 12 or 16 MB of cache such as Intel's Xeon 7400 series.

5.1.3 Using smaller tables

In order to avoid L2/L3 cache pressure, the Whirlwind round transformation can also be implemented using smaller lookup tables. In these cases, γ has to be implemented by separate table lookups requiring a table of $2^{16} \cdot 2$ bytes. Since the λ_i are applied independently, the approach described in equation (9) can be generalized to lookup tables for one, two or four λ_i mapping scalars of 4, 8 or 16 bits to complete 128-bit rows with the individual contributions shifted to the final location. Then an output row can be obtained by XOR-ing 4, 2 or 1 partial rows together eight times. The best trade-off for contemporary machines is offered by combining two λ_i per table. In total, this implementation needs $2^{17} + 2^{16}$ bytes (192 kB) of lookup tables, which easily fits in most L2 caches.

As seen in Table 4, this implementation ("medium lookup tables"), albeit needing three times as many lookups per round transformation as in the big tables approach, improves performance especially on CPUs with smaller cache size. The speedup this implementation receives on the machines with bigger cache sizes are explained by them also featuring more recent microarchitectures.

Table 4 Performance figures for software implementations of Whirlwind. All numbers are given in cycles per byte (cpb)

	Intel Xeon E5540 2.53 GHz, 8 MB L3 cache (cpb)	Intel Core 2 X9650 3 GHz, 6 MB L2 cache (cpb)	Intel Xeon E5335 2 GHz, 4 MB L2 cache (cpb)
Large lookup tables	151.31	210.32	217.86
Compressed inverse	129.43	135.61	207.74
Medium lookup tables	99.58	124.35	145.18
Bitsliced (2 blocks)	63.22	63.14	67.79
Whirlpool	54.60	38.15	56.31

5.1.4 Using the symmetry of the inverse

The use of normal bases in Whirlwind permits another implementation variant based on the following symmetry property: If $(a, b)^{-1} = (c, d)$, then $(b, a)^{-1} = (d, c)$ and $(a, a)^{-1} = (c, c)$. This can be directly verified by Eq. 1 and suggests the following procedure for computing the inverse of (a, b) in GF(2^{4m}):

$$(a,b)^{-1} = \begin{cases} S[a,b] & \text{if } a < b, \\ (S[a,a], S[a,a]) & \text{if } a = b, \\ S[b,a] & \text{if } a > b; \end{cases}$$

with a table S storing only the inverses (c, d) of (a, b) with $a < b$ and the value c for the inverse of $(a, a)^{-1} = (c, c)$. This table then only requires $2^{2m} \cdot (2^{2m} - 1)/2 \cdot 4m + 2m \cdot 2^{2m} = 2m \cdot 2^{4m}$ bits of memory, which is half the size of the full lookup table. For Whirlwind, $m = 4$, so this implies a table of 64 kB instead of 128 kB.

However, this reduction comes at the expense of conditional processing. So while this technique can also be used to halve the size of the tables described by (9), the cost of the additional processing practically compensates for this, as seen in Table 4. On the other hand, table-based implementations on platforms where memory is the primary concern will benefit from the reduction provided by this technique (see also Sect. 5.2).

5.1.5 Bitslicing

In order to obtain a constant-time implementation resistant to side-channel attacks, Whirlwind can also be implemented in a bitsliced manner. Similar to the recent very fast bitsliced implementation of the AES [17], the normal basis decomposition of the field arithmetic presented in Sect. 3 leads to compact formulations on the level of individual bits. The current implementation is using Intel's SSE3 instruction set and processes two blocks of two independent hashing operations in parallel to fully utilize the register width. Improving this implementation and deriving a representation for single block hashing that still leads to efficient computations is currently work in progress.

5.2 Embedded platforms and hardware

At the moment, there are no implementations for 8-bit processors or in hardware. However, by making use of the tower field decomposition employed in Whirlwind, the implementation techniques described in [24] can be applied. Also, we expect the compactness estimations from this paper to carry over proportionally, resulting in competitive implementations for resource-constrained platforms.

5.2.1 Memory requirements on 8-bit platforms

For implementations on 8-bit microcontrollers, the requirements in terms of RAM and ROM are generally a far greater concern than execution speed. We estimated those requirements for Whirlwind according to the criteria mentioned in [14], which in particular implies that the IV is stored in ROM and that the memory used for the message block is considered external to the hash algorithm and hence not taken into account.

An implementation of Whirlwind needs 512 bits of ROM for the IV and at least 512 bits of RAM to store the previous chaining value for the feed-forward and 1,024 bits of RAM for the internal state. The round constants can either be implemented using 768 bits of ROM

or via a counter using 8 bits of RAM. As described in Sects. 3.3 and 3.4, the finite field arithmetic can be recursively decomposed down to binary operations. While this comes at a performance penalty on 8-bit platforms, it eliminates the need for lookup tables. Alternatively, the technique from Sect. 5.1.4 can be used to speed up the implementation at the cost of 2^{16} bytes of ROM for a lookup table of the compressed inverse. Combined approaches are possible, for instance using the normal basis arithmetic for the first level of decomposition and then employing lookup tables for the smaller subfields.

Summarising, we estimate that Whirlwind can be implemented on 8-bit platforms using either 192 bytes of RAM and 160 bytes of ROM, or 193 bytes of RAM and 64 bytes of ROM. Compared to the SHA-3 candidates analyzed in [14], this places Whirlwind in the "Middle" class of algorithms with regard to memory requirements, for instance being significantly smaller than the Round 2 candidates ECHO and SIMD, comparable to BMW, Shabal and Keccak, and larger than BLAKE, Hamsi and Luffa.

6 Conclusion

Whirlwindis a hash function based on and improving on the Whirlpool design. It employs large S-boxes (16 bits) which still allow efficient and flexible implementations on a wide variety of platforms due to the normal bases decomposition. During the design, recent refinements of the understanding of the wide-trail strategy have been taken into account, improving the differential behavior and the complexity of algebraic descriptions. Arguments are provided for resistance to attacks, including conventional differential cryptanalysis and recent improvements, in particular the rebound attack.

In comparison to the algorithms submitted to the SHA-3 competition, Whirlwind takes recent developments in cryptanalysis into account *by design*. While current software implementations are not exceptionally fast, they compare favourably with the 512-bit versions of SHA-3 candidates such as LANE [16] or the original CubeHash proposal [4] and are about on par with ECHO [3] and MD6 [28].

Acknowledgments We would like to thank the referees for their comments which improved the paper. This work was sponsored by the Research Fund K. U. Leuven, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the European Commission through the ICT Programme under Contract ICT-2007-216676 (ECRYPT II). Elmar Tischhauser is a research assistant of the F.W.O., Fund for Scientific Research—Flanders.

References

1. Barreto P., Rijmen V.: The Anubis block cipher. First open NESSIE Workshop, Leuven, November 13–14 (2000).
2. Barreto P., Rijmen V.: The Whirlpool hashing function. First open NESSIE Workshop, Leuven, November 13–14 (2000).
3. Benadjila R., Billet O., Gilbert H., Macario-Rat G., Peyrin T., Robshaw M., Seurin Y.: SHA-3 Proposal: ECHO. Submitted to NIST (2008).
4. Bernstein D.J.: CubeHash Specification. Submitted to NIST (2008).
5. Bertoni G., Daemen J., Peeters M., Van Assche G.: On the Indifferentiability of the Sponge Construction. EUROCRYPT, LNCS, vol. 4965, pp. 181–197 (2008).

6. Biham E., Dunkelman O.: The SHAvite-3 Hash Function. Submitted to NIST (2008).
7. Biryukov A.: Design of a New Stream Cipher—LEX. New Stream Cipher Designs, LNCS, vol. 4986, pp. 48–56 (2008).
8. Contini S., Lenstra A.K., Steinfeld R.: VSH, an Efficient and Provable Collision-Resistant Hash Function. EUROCRYPT, LNCS, vol. 4004, pp. 165–182 (2006).
9. Daemen J., Rijmen V.: The Design of Rijndael: AES—The Advanced Encryption Standard. Springer (2002).
10. Daemen J., Rijmen V.: Plateau characteristics and AES. IET Inf. Secur. **1**(1), March 2007, 11–17.
11. Daemen J., Rijmen V.: New criteria for linear maps in AES-like ciphers. Cryptography and Communications Discrete Structures, Boolean Functions and Sequences, vol. 1, no. 1. Springer, pp. 47–69 (2009).
12. Gauravaram P., Knudsen L.R., Matusiewicz K., Mendel F. Rechberger C., Schläffer M., Thomsen S.S.: Grøstl—a SHA-3 Candidate. Submitted to NIST (2008).
13. Hilewitz Y., Yin Y., Lee R.: Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation. FSE, LNCS, vol. 5086, pp. 173–188 (2008).
14. Ideguchi K., Owada T., Yoshida H.: A Study on RAM Requirements of Various SHA-3 Candidates on Low-cost 8-bit CPUs. May 2009. http://www.sdl.hitachi.co.jp/crypto/lesamnta/A_Study_on_RAM_Requirements.pdf.
15. IEEE 1363 draft 13: Standard Specifications for Public Key Cryptography, November 1999. http://grouper.ieee.org/groups/1363/.
16. Indesteege S.: The LANE Hash Function. Submitted to NIST (2008).
17. Käsper E., Schwabe P.: Faster and Timing-Attack Resistant AES-GCM. CHES, LNCS, vol. 5747, pp. 1–17 (2009).
18. Lamberger M., Mendel F., Rechberger C., Rijmen V., Schläffer M.: Rebound Distinguishers: Results on the Full Whirlpool Compression Function. ASIACRYPT, LNCS, vol. 5912, pp. 126–143 (2009).
19. Lidl R., Niederreiter H.: Introduction to Finite Fields and Their Applications. Cambridge University Press, London (1986).
20. Matusiewicz K., Naya-Plasencia M., Nikolic I., Sasaki Y., Schläffer M.: Rebound Attack on the Full LANE Compression Function. ASIACRYPT, LNCS, vol. 5912, pp. 106–125 (2009).
21. Mendel F., Rechberger C., Schläffer M., Thomsen S.: The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. FSE, LNCS, vol. 5665, pp. 260–276 (2009).
22. Mullin R., Onyszchuk L., Vanstone S., Wilson R.: Optimal Normal Bases in GF(p^n). Discr. Appl. Math. **22**(2), 149–161 (1989).
23. Nakajima J., Matsui M.: Performance Analysis and Parallel Implementation of Dedicated Hash Functions. EUROCRYPT, LNCS, vol. 2332, pp. 165–180 (2002).
24. Nikova S., Rijmen V., Schläffer M.: Using Normal Bases for Compact Hardware Implementations of the AES S-Box. SCN, LNCS, vol. 5229, pp. 236–245 (2008).
25. Nyberg K.: Differentially uniform mappings for cryptography. EUROCRYPT, LNCS, vol. 765, pp. 55–64 (1992).
26. Paar C.: Efficient VLSI Architectres for Bit-Parallel Computations in Galois Fields. Ph.D. thesis, University of Essen (1994).
27. Perlis S.: Normal bases of cyclic fields of prime-power degree. Duke Math. J. **9**(3), 507–517 (1942).
28. Rivest R.L.: The MD6 Hash Function—A Proposal to NIST for SHA-3. Submitted to NIST (2008).
29. Saarinen M.-J.O.: Security of VSH in the Real World. INDOCRYPT, LNCS, vol. 4329, pp. 95–103 (2006).
30. Vaudenay S.: Hidden Collisions on DSS. CRYPTO, LNCS, vol. 1109 pp. 83–88 (1996).